LAWOW: Lightweight Android Workload Offloading Based on WebAssembly in Heterogeneous Edge Computing

1st Shu Zhu
College of Computer
National University of Defense Technology
Changsha, China
zhushu97@126.com

2nd Bao Li*
College of Computer
National University of Defense Technology
Changsha, China
baoli@nudt.edu.cn

3rd Yusong Tan
College of Computer
National University of Defense Technology
Changsha, China
tys@tys.zone

4th Xiaochuan Wang
College of Computer
National University of Defense Technology
Changsha, China
wangxiaochuan@yhkylin.cn

5th Jianfeng Zhang
College of Computer
National University of Defense Technology
Changsha, China
zhangjianfeng@yhkylin.cn

***Abstract*—In recent years, the rapid growth of IoT and AI has placed higher demands on computing power. Edge computing extends cloud computing to edge devices by increasing the resource storage capacity of network edge nodes as well as computing power, enabling edge devices to process the request tasks of user devices closer to users, thus reducing response time and providing better services to users. Edge computing offloads the tasks that need to be processed to edge servers to reduce latency and compute costs. Compared to the public cloud, Existing offloading approaches based on virtual machines and containers incur disadvantages of slow running and large memory footprint, which will lead to the loss of the advantages gained in deploying tasks on the edge. In this paper, we propose LAWOW, a lightweight and high-performance framework to support task offloading using wasmedge, a generic runtime for WebAssembly. We aim to take advantage of the cross-platform nature of wasmedge to enable offloading workloads from android systems to the edge server without the need to re-develop and deploy the applications according to the platform. LAWOW directly offloads key WebAssembly files to run on the edge server. Experiment results show that LAWOW can reduce the computation cost while with smaller memory footprint.**

***Keywords*-Android Workload; Offloading; WebAssembly; Edge Computing;**

I. Introduction

In recent years, the rapid development of mobile terminal devices (e.g., smartphones, laptops) and wireless communication technologies has led to exponential growth in data scale. The rich application scenarios brought by these devices, such as Smart Homes, autonomous driving, virtual reality, etc., also challenge the existing computing power and storage resources. As an emerging paradigm, Edge Computing improves the efficiency of overall computing task implementation by providing low-latency, low-energy and highly flexible services to users. Edge computing refers to moving the location of processing tasks from the cloud computing center to the edge nodes closer to the user's device and providing certain computing power and network resources to the edge nodes. The main purpose is to try to reduce the distance to make the data processing closer to the user, thus reducing the problems such as high latency and low privacy due to the long distance.In order to perform resource-intensive tasks of user devices in edge servers, user devices offload some tasks to edge servers for computational processing and pass the obtained results back to user devices, thus increasing the speed of ~~task~~ execution and reducing the energy consumption of the devices. At this stage, most of the research on computational offloading focuses on optimizing algorithms for computational offloading decisions and the design of offloading frameworks[1].

In the past, task offloading mainly involved offloading tasks to remote cloud computing service centers, and high cost, high latency and privacy issues can be solved by offloading tasks to edge servers near users instead of remote data centers. However, On the other hand, the computation, storage and network resources in the edge computing scenario are too limited compared with cloud data centers[2].

979-8-3503-3293-3/22/$31.00 ©2022 IEEE
DOI 10.1109/ISCTech58360.2022.00124

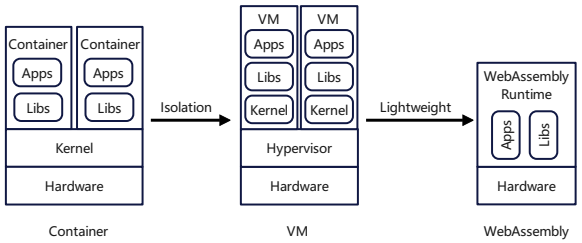


Figure 1: Differences among container, VM and WebAssembly.

At the same time, the edge also need to meet the needs of different architectures and applications, which puts higher requirements on the efficient scheduling and optimization of edge resources. Virtualization, as the main way to achieve resource abstraction, is widely used in cloud data centers, such as virtual machines (VMs), containers[3] and so on. VMs occupy huge memory and have long startup time. Meanwhile, containers occupy smaller space and require less resources than VMs, the startup time is still too long and the isolation is weaker than VMs with independent operating systems. In public cloud environments, container applications have to run in virtual machines to meet multi-tenant security and isolation requirements, which runs counter to the flexible and lightweight nature of containers. There is waste in resource utilization and operational efficiency.

In this paper, we leverage WebAssembly, an emerging binary instruction format for a stack-based virtual machine, to design an offloading framework for android workloads. WebAssembly can combine isolation, lightweight, requirements and ignore the challenges of underlying hardware device heterogeneity compared to VMs and containers. Figure 1 shows the difference of VM, container and WebAssembly[4]. Meanwhile, we choose wasmedge, which is the fastest WebAssembly runtime available today, as the lightweight, high-performance and scalable WebAssembly runtime, in our design. There are still some difficulties to overcome when using wasmedge as a runtime to handle offload requests in edge computing, such as mobile code support, offload code conversion, etc. In this paper, we design an offload model for Android applications using the wasmedge runtime. The model aims to generate apk files using WebAssembly bytecode files packaged so that the duplicate bytecode files can run on both the android system and the edge server without multiple re-compilations. The offload model contains three parts: packer, classifier, and executor. By embedding the WebAssembly code into the application code for packaging, then classifying and reconstructing the parts to be offloaded on the user device, and finally processing and executing the program on the offload server to reduce latency.

The main idea and contributions of this paper could be summarized as follows.

- We propose the idea of using WebAssembly in android workload, which can be an effective solution for the heterogeneity of edge computing.
- We designed a new offload model based on the fastest runtime——wasmedge, which allows the workloads to be migrated directly across platforms.
- Our proposed offload model is more lightweight in terms of memory footprint, energy consumption, and latency.It also better fits the resource-constrained environment of edge devices.

The rest of this paper is organized as follows. Section 2 introduces the current state of research and related work, Section 3 introduces the concept of WebAssembly and the design and implementation of wasmedge-based offloading framework, and in Section 5, we evaluate the work based on wasmedge runtime and compare the performance and resource consumption with VMs and container. Finally, we conclude the paper with a discussion of the limitations of the approach and an outlook for the future.

II. Related Works

Task offloading is now a popular research direction in edge computing, which focuses on reducing service latency and device energy consumption by partially or fully processing compute-intensive tasks on some user devices at the network edge. Recently, many researchers have made significant contributions in this area. The ThinkAir[5] framework enhance cloud computing by using multiple virtual machine images to perform tasks in parallel, thereby reducing computational latency and getting offload results quickly. CloneCloud[6] uses an application partitioner to partition the mobile application and migrates the offloaded portion running in the virtual machine to the cloud device, processing it by executing the runtime. The DPartner[7] framework can automatically refactor Android applications to have computational offload capabilities, analyze Android program bytecode, and rewrite it to support special program structures for on-demand offloading.

Another task offloading research area focuses on optimizing offloading decision algorithms. Li et al. proposed a deep reinforcement learning-based resource allocation strategy for offloading schemes in vehicular networks, which solves with the optimization objectives of shortest latency and minimum computational cost using Q-learning[8]. Yang et al. describe the task offloading problem between UAVs as an integer programming problem and propose a new offloading model as well as a game strategy to solve the above problem.[9]. Chen[10] jointly optimizes the offloading decision and communication resource allocation by representing this optimization as a mixed integer programming problem and transforming it into a non-convex quadratically constrained quadratic programming (QCQP) problem.

It can be seen that most of the research at this stage focuses on offloading decision optimization and framework design. As most of the research mainly focuses on the implementation details of the offloading process, we often ignore the huge burden of the underlying heterogeneity and environment configuration to the edge servers. In order to further reduce latency and energy consumption, some of the research considers lightweight virtualized containers and runtime. Wang et al. designed and implemented a framework and approach for mobile cloud computing offloading based on Docker containers[10]. Rattrap[11] used LXC (Linux container) as an offload for cloud server runtime. The container environment significantly reduces the cloud overhead compared to VM. In contrast to VM-based and container offloading mechanisms that rely on hardware virtualization techniques, the WebAssembly runtime used in this paper mainly converts applications into platform independent bytecode, thus ignoring the underlying hardware device differences.

III. System Design

WebAssembly is a portable, small memory footprint, fast-loading emerging bytecode format developed intially to partially replace JavaScript and accelerate the execution of application code in the browser by compiling the code into WebAssembly bytecode so that it runs in the browser at near-native speeds. Due to its portability and the security of a sandboxed execution environment, it has been extended to the server side and considered as a promising alternative to container technology[12]. Compared to VMs and containers, WebAssembly has the advantages of being small, fast and secure. The reason it can be compiled only once to run anywhere runtime exists is that it can ignore the computer hardware architecture on which it runs and provides a WebAssembly system interface (WASI). WASI is an interface that can be ported across WebAssembly programs and OS kernels, making WebAssembly bytecode to run smoothly on heterogeneous platforms. In order to run WebAssembly in edge or cloud environments, a number of new WebAssembly runtimes have been introduced, which provide libraries to convert applications to WebAssembly modules in any programming language. The runtime used in this paper is wasmedge, a lightweight, high-performance and scalable WebAssembly runtime and the fastest WebAssembly VM available today, which provides an execution sandbox environment for WebAssembly bytecode programs in providing isolation and protection from OS resources and memory space. In this paper, we embed the wasmedge runtime into the task offload framework to convert Android application code to WebAssembly bytecode and exploit its portability to offload to edge servers without recompiling for multiple classes of device architectures to run at near-native speeds, thus alleviating the problem of insufficient device resources and high latency.

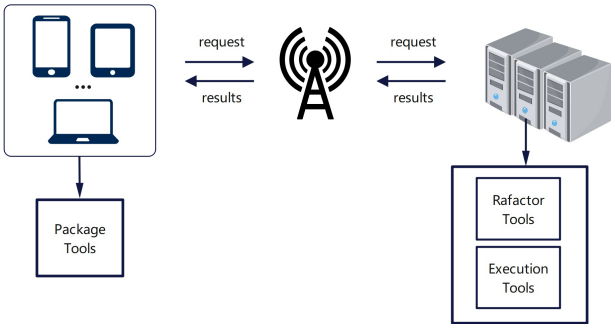


Figure 2: System Architecture

Wasmedge, while having many advantages over vm and containers, still has some challenges as a runtime for mobile application offload. Firstly because we are executing a WebAssebly bytecode program, we need to embed that bytecode program into the android application. Secondly, an android application is essentially a java program implemented by constructing classes through a series of computational functions, so the uninstallation process can be achieved by sorting the classes in the application and selecting its key WebAssembly classes. Some of the files it depends on for remote deployment[13]. Finally, to run the offloaded files on the edge devices, a module needs to be built to store the dependencies and library packages needed to offload the server to execute the files. Figure 2 shows the system architecture. The above process is divided into three main parts: Package Tools, Refactor Tools, and Execuation Tools, which will be described below.

Package Tools: The version of the wasmedge runtime comes with a pre-purchased build binary for the android operating system. The wasmedge source code is built into the android shared library file via a cmakelist file, and the wasmedge runtime is embedded into the final apk application. Taking the fib function as an example, the android ui application accesses the wasmedge function by calling the kotlin object, which uses the Java Native Interface (JNI) to load the called C shared library. Then executes the WebAssembly bytecode by calling the Wasmedge C SDK embedded in the shared library into the Wasmedge VM. The process is shown in Figure 3

Refactor Tools: We borrowed the idea of DPartner[7] refactoring tool for refactoring Android applications, which first analyzes the classes in the program that need to be offloaded and rewrites the application to generate two parts to be deployed on user devices and offload servers.As Figure4 shows it consists of the following two steps.

- Step 1: In packaged tools we have converted the main function implementation functions of the application into WebAssembly bytecode, called using the wasmedge function and saved in the corresponding WebAssembly class with the corresponding naming. So

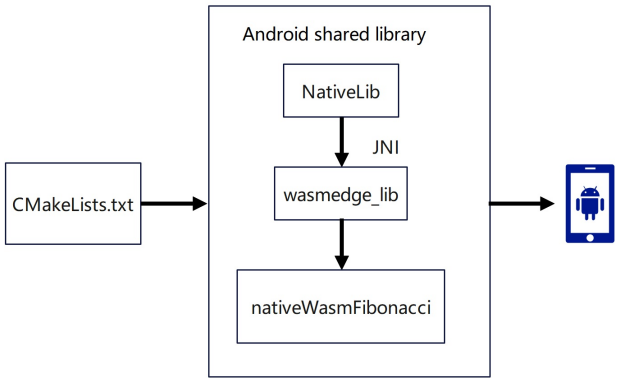


Figure 3: Building process of Package Tools.

for a given application, the refactor tools first step will automatically divide the application into two classes by: the WebAssembly class that has been specially named and other classes that are unique resources that support applications running on smartphones, such as GUI displays, etc.

- Step 2: The input to the Refactor Tools is the java bytecode file of the Android application and the referenced resource files, such as images and xml files. These files are packaged into two files after differentiation based on class names, one is the refactored Android application, i.e. apk file, and the other is an executable jar file containing the webassembly bytecode file that needs to be uninstalled.

Execuation Tools: After extracting the core WebAssembly bytecode file from the application separately via the WebAssembly extractor and offloading it to the edge server, we use the wasmedge runtime to execute the bytecode. The most significant advantage of using WebAssembly is that it ignores the underlying hardware heterogeneity, which means that the underlying architecture does not have to be the same for application development and deployment. Once a WebAssembly bytecode program is created, it can be deployed anywhere WebAssembly is supported. So we do not need to recompile the core bytecode of the application on the Android device to the edge server again according to the underlying architecture of the server during the offloading process, we just need to run the same bytecode directly. Therefore, we design functions in the server to run interrupt programs to execute the bytecode. For example, we use the command wasmedge –reactor fibonacci.wasm fib 10 to execute the Fibonacci function with argument 10.

IV. Evaluation

A. Experiment setup

In this section, we evaluate LAWOW on a PC as an edge server. The PC configuration is a dual-core Intel(R)core(TM) i9-10900 2.80GHz processor with 32GB RAM, 512GB SSD, running Ubuntu 20.04 LTS. For end device we choose HUAWEI Mate 10 which consists of HiSilicon Kirin 970 and 4GB memory. For the offloading framework we choose ThinkAir[5], for the classifier we borrowed the classification idea from DPartner[7] and modified it according to our own runtime.We compare the following two scenarios, one is running the application on the Android local device and the other is running the application after offloading it to the edge server.

The metrics of our setup are as follows.

- **Time,** mainly contains the processing time of the application, where the LAWOW processing time includes the transmission time of the program and the execution time on the edge server, while the Android processing time is the execution time of the application locally on the user's device.
- **Memory,** the main measurements are the memory usage of the application when using runtime processing and the memory usage of the application running with VM and container. The VM uses Ubuntu 20.04 LTS running in vmware workstation. The container uses Rattrap[11], which runs a virtual Android runtime in LXC by extending OS kernel with Android drivers on demand[4].

For the offloaded assembly, Table I shows the set of three different types of computational loads that we have used. These tests were done using WiFi.

Table I: Benchmark Programs

Program	n	notes
Matrix Multiply	500	(n,n)*(n,n)
Fibonacci	32	
Insertion Sort	30000	linear search

B. Results and discussion

- **Time:** We compare the execution time of an application running locally on an android phone with the execution time of an application offloaded to an edge server using the offload model we designed. The execution time is divided into three main components: data transfer, runtime preparation, and program runtime. As can be observed from Figure 5(a), For three types of computational intensive applications, the execution time of LAWOW is reduced by 84% compared to that executed locally when processing the insert_sort computation task, while the execution time of the matrix-multiply is reduced by 90% and the execution time of the Fibonacci sequence algorithm is reduced by 73%. We can conclude that LAWOW is more efficient in terms of reducing task processing time.
- **Memory:** The main system overhead we focus on is memory footprint[4], which is an important metric for

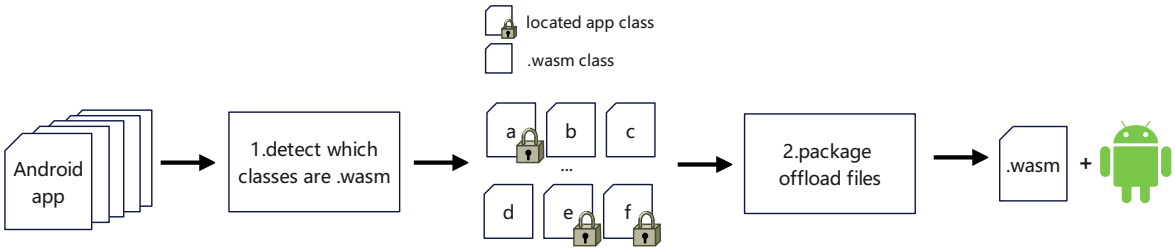
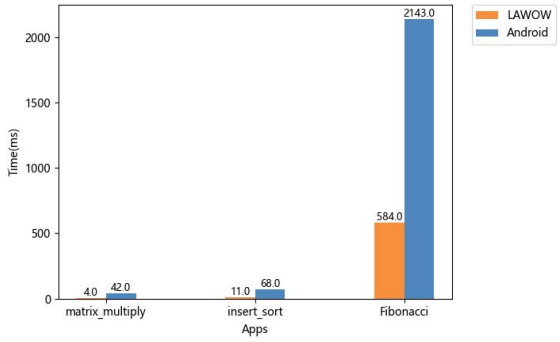
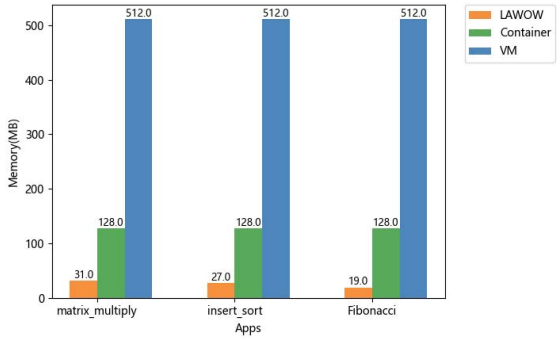


Figure 4: Refactor Tools workflow.



(a) Comparison of time



(b) Comparison of memory

Figure 5: (a) and (b) show the time and memory comparison between the three types of programs we tested running natively on the Android platform and in the LAWOW.

evaluating systems on edge devices. Less memory footprint means we can launch more applications for the same cost of physical resources. Figure 5(b) shows that VM requires at least 512MB of memory to start while containers need 128MB. However, when we offload the application to the PC and run it with wasmedge it only takes less than 50MB of memory to process the program. In terms of memory footprint, the wasmedge-based runtime is approximately 90% lower than the traditional VM-based runtime. As you can see, the VM image is too large and contains the full operating system, so it consumes too much in boot time and energy. Containers perform slightly better compared to VMs, but LAWOW is still the best performer in terms of memory and energy consumption.

V. CONCLUSION AND FUTURE WORK

In this paper, we introduce LAWOW a WebAssembly based android workload offloading framework to achieve further latency and resource overhead reduction, while ignoring the heterogeneity of edge servers in edge computing. Our evaluation results show that LAWOW performs better in terms of system overhead and execution time compared to the traditional VM and container environment.

Since currently we only evaluate LAWOW via specific workloads on Android phones, in the future we will extend our work to more edge scenarios, such as complex applications which include intensive AI computation steps to improve the generality.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China with grant number U19A2060.

REFERENCES

[1] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[2] A. Islam, A. Debnath, M. Ghose, and S. Chakraborty, "A survey on task offloading in multi-access edge computing," *Journal of Systems Architecture*, vol. 118, p. 102225, 2021.

[3] W. Huaijun, T. Ling, L. Junhuai, and G. Zhe, "Research and implementation of mobile cloud computing offloading system based on docker container," in *2017 10th International Symposium on Computational Intelligence and Design (ISCID)*, vol. 2, 2017, pp. 270–274.

[4] S. Wu, C. Mei, H. Jin, and D. Wang, "Android unikernel: Gearing mobile code offloading towards edge computing," *Future Generation Computer Systems*, vol. 86, pp. 694–703, 2018.

[5] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 945–953.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*, 2011, p. 301–314.

[7] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android java code for on-demand computation offloading," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012, p. 233–248.

[8] X. Li, "A computing offloading resource allocation scheme using deep reinforcement learning in mobile edge computing systems," *J. Grid Comput.*, vol. 19, no. 3, sep 2021.

[9] Y. Li, B. Yang, H. Wu, Q. Han, C. Chen, and X. Guan, "Joint offloading decision and resource allocation for vehicular fog-edge computing networks: A contract-stackelberg approach," *IEEE Internet of Things Journal*, vol. 9, no. 17, pp. 15 969–15 982, 2022.

[10] M.-H. Chen, M. Dong, and B. Liang, "Resource sharing of a computing access point for multi-user mobile cloud offloading with delay constraints," *IEEE Transactions on Mobile Computing*, vol. 17, no. 12, pp. 2868–2881, 2018.

[11] S. Wu, C. Niu, J. Rao, H. Jin, and X. Dai, "Container-based cloud platform for mobile computation offloading," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 123–132.

[12] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "Webassembly modules as lightweight containers for liquid iot applications," in *Web Engineering*, 2021, pp. 328–336.

[13] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 140–149.